VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2016 Richard Hlavica

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Richard Hlavica**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: TELE DATA SYSTEM, spol. s r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Petr Olivka, Ph.D.**

Konzultant bakalářské práce: Bc. Adam Lindovský

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 29. dubna 2016

.....

TDS Tvůj Datový Systém spol. s r.o. 1
28. října 1418/125, 702 00 Mor. Ostrava
DIČ: CZ18051286

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, hlavně kolegům z mého týmu, kteří mi byli vždy nápomocni při zadaných úkolech.

Abstrakt

Obsah této bakalářské práce popisuje mou odbornou praxi ve firmě Tele Data System s.r.o. V průběhu praxe jsem byl začleněn do týmu vývojářů, kteří programují v programovacím jazyce Java. Mým hlavním úkolem bylo vylepšovat a opravovat již vytvořenou aplikaci, která se neustále vyvíjela. Hlavně jsem používal programovací jazyk Java, ale také i technologii JUnit a RCP.

Klíčová slova: Tele Data System, TDS, Java, RCP, JUnit, Bakalářská praxe

Abstract

Main contain of my bachelor thesis explain my professional practice in the company Tele Data System s.r.o. I was included to the team of developers who uses Java programming language. My main task was to improve and refactor already created application which was constantly evolved. Mostly I was programming with Java but also with JUnit and RCP technology.

Key Words: Tele Data System, TDS, Java, RCP, JUnit, bachelor practice

Obsah

Seznam použitých zkratek a symbolů

RCP	–	Rich Client Platform
TDS	–	Tele Data System s.r.o.
TDE	–	TELE DATA ELECTRONIC GmbH
SCADA	–	Supervisory Control And Data Acquisition
HTML	–	HyperText Markup Language
XML	–	Extensible Markup Language
API	–	Application Programming Interface
PFS	–	Product Feature Specification
SVN	–	Subversion
CSV	–	Comma-separated values
IP	–	Internet Protocol

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Závěrečnou práci bakalářského studia jsem se rozhodl udělat formou praxe ve firmě Tele Data System s.r.o. Rozhodl jsem se tak, jelikož si myslím, že tato forma bakalářské práce bude pro mne zajímavější a přinese mi nové zkušenosti a odborné znalosti.

Firmu TDS jsem si vybral proto, že se v ní programuje v jazyce Java, který se mi během studia nejvíce zalíbil a myslím si, že bude stejně rozšířený za několik let jako dnes. Další důvod pro můj výběr firmy byl ten, že jsem mohl pracovat v týmu až pěti lidí, což je pro mne mnohem přijatelnější, než kdybych pracoval sám.

V následujících kapitolách popíši firmu, kde jsem pracoval, co jsem měl za úkoly a jaké nové zkušenosti a odborné znalosti jsem získal.

2 Popis odborného zaměření firmy a popis pracovního zařazení studenta

2.1 Historie firmy TDS

Firma TDS, jako sesterská firma německé společnosti TELE DATA ELECTRONIC GmbH, vznikla roku 1991. Firma se od svého vzniku zabývala realizací telemetrických a dispečerských systémů především ve vodárenských společnostech. Pro budování těchto systémů byly využívány produkty mateřské firmy TDE. O rok později byl ve firmě dokončen vývoj SCADA systému, který byl provozován víceuživatelským operačním systémem MultiUser-DOS.

Roku 1997 byla realizována a předána do provozu největší zakázka v oblasti čistíren odpadních vod pro Ústřední čistírnu odpadních vod Ostrava.

Další velký projekt byl pro Severomoravskou plynárenskou a.s. realizace systému pro dálkové řízení předávací stanice plynu. Dále byla do provozu uvedena aplikace v oblasti energetiky pro jadernou elektrárnu Dukovany, kde bylo do provozu uvedeno 13 měřících ústředen rychlých binárních signálů [5].

Také nutno podotknout, že se firma zabývá i veřejně prospěšnou činností ve formě příspěvků různým klubům a společnostem.

2.2 Popis pracovního zařazení studenta

Byl jsem zařazen do týmu Acos ET. Acos je aplikace, která se neustále vyvíjí a udržuje.

V době mého přijetí měl tým 4 členy, 3 programátoři, které vede jeden team-leader, a jeden tester. Vždy, když se dokončil nějaký úkol, tak se musel otestovat právě testerem.

Jak už jsem se zmínil v úvodu, programoval jsem v jazyce Java. Aplikace Acos je z větší části naprogramovaná právě v tomto jazyce. Používá se zde také i HTML, Groovy a XML.

Úkolem mé práce není jenom implementace oprav nebo nového kódu, ale také testování všech tříd. Pro testování jsem používal JUnit a nejčastěji balíček Mockito.

Pokaždé, když jsem dokončil implementaci kódu a testovacích tříd, založil jsem code-review. Tam jsem přidával všechny mé změny a všichni kolegové se na ně mohli podívat, případně napsat komentář, co bych měl udělat lépe. Pro tyto kontroly jsem používal nástroj Crucible.

Úkoly jsem dostával prostřednictvím softwarového nástroje JIRA.

3 Nástroje a technologie používané při práci

K práci jsem dostal pracovní notebook, na který jsem si nainstaloval všechny potřebné aplikace a nastavil všechna nezbytná nastavení pro komunikaci se servery v Německu. K notebooku mi také dali druhý monitor, takže práce pro mě byla rychlejší a přehlednější.

3.1 Nástroje

Občas jsem se potřeboval domlouvat na implementacích s kolegy v Německu. Nejčastěji komunikace probíhala prostřednictvím aplikace Skype, kde je vytvořená skupina, ve které jsme všichni z týmu z České Republiky a všichni z německého týmu. Pokud se chystala razantní změna do aplikace, tak jsem dostal e-mail s PFS. To je dokument, který obsahoval popis, jakou změnu jsme měli implementovat a jaké měla mít chování.

Jako API jsem používal Eclipse se kterým jsem se setkal při studiu, takže jsem si nemusel zvykat na nové prostředí. Ovšem na praxi jsem se dozvěděl plno jiných možností a vlastností, které tento program má. V Eclipse jsem používal pro verzování kódu SVN. To je systém, pro správu verzí a zdrojových kódů.

Jako druhý program pro správu verzí jsem používal TortoiseSVN. To je podobné jako SVN v Eclipse, akorát výhoda je ta, že není závislý na API, takže se mohou synchronizovat soubory přímo v průzkumníku což bylo většinou rychlejší. Také se přes tento program daly synchronizovat jiné soubory, které v Eclipse nebyly viditelné. Stávalo se, že bylo nutné synchronizovat přes TortoiseSVN a ne přes Eclipse.

Jelikož Acos projekt může být přístupný i pro více uživatelů, bylo nutné tyto projekty nahrát na VisualSVN server, ke kterému jsme se potom připojili a testovali chování aplikace při práci více uživatelů najednou. Tam jsem si celý projekt uložil a vytvořil si několik uživatelů, kteří na tento projekt mohli přistupovat.

3.2 Technologie

Hlavním programovacím jazykem byla Java [2]. Java je jedním z celosvětově nejrozšířenějších programovacích jazyků. Na rozdíl od jiných programovacích jazyků si toto postavení udržuje již několik let a její vliv pořád vzrůstá. Hlavními důvody úspěchů Javy jsou její pružnost a přizpůsobivost. Také se podílí na celosvětové revoluci, kterou způsobily chytré mobilní telefony, neboť se používá pro programování aplikací pro systém Android.

RCP je stand-alone aplikace založená na Eclipse platformách. Celý Eclipse je vlastně RCP aplikace [6].

JUnit [7] je jednoduchý framework pro psaní testů. JUnit testy se píší v jazyce Java.

XML je obecný značkovací jazyk. Při práci jsem ho používal hlavně pro model celého Acos programu.

MAVEN je nástroj pro správu a řízení sestavení aplikací.

4 Zadané úkoly

Zadaných úkolů jsem měl mnoho. Některé byly na kratší dobu, některé na delší. Ze začátku, když jsem nastoupil na praxi, jsem se ve všech třídách moc nevyznal, a tak mi implementace různých úkolů trvala delší dobu.

Postupem času, jak jsem získával přehled o většině pomocných tříd, se čas na implementaci zkrátil. Vždy, když jsem dostal úkol, bylo u něj napsáno, jaká je jeho předpokládaná časová náročnost. Některé úkoly byly tak velké, že se musely rozdělit na více menších úkolů a tím pádem na tom mohl pracovat celý tým. Časová náročnost byla většinou v rozmezí od dvou hodin až po dvacet dní.

Měl jsem i úkoly, které jsem měl hotové dříve. Někdy jsem se však do předpokládané doby implementace nevešel a pracoval jsem na úkolech déle, protože se většinou vyskytly skryté problémy.

V následujících kapitolách popíšu přípravu a konečnou implementaci časově náročných úkolů, ale i pár méně časově náročných.

4.1 Seznámení s aplikací Acos

Acos je velmi rozsáhlá aplikace. Její model má přes dvacet pět tisíc řádků.

Po spuštění aplikace uživatel mohl otevřít nebo vytvořit nový projekt. Projekt je jako objekt ve stromové struktuře hierarchicky nejvýše. Projektů v aplikaci jsem mohl mít otevřených víc, avšak nesměly to být zabezpečené nebo víceuživatelské projekty. Takových projektů v aplikaci nesmělo být otevřených více než jeden.

Pracoval jsem tedy se třemi různými druhy projektů. Dva poslední zmíněné jsou zabezpečené. Do takových projektů bylo možné se přihlásit pomocí uživatelského jména a hesla.

V projektu se vytvářely různá zařízení. Zařízení existuje celá řada, ale mají společné některé atributy. Třeba unikátní identifikační číslo, adresa zařízení, jméno zařízení, rodiče a tak dále. Rodič zařízení je projekt.

Zařízení se mohly spojovat pomocí různých komunikací a mohly se na nich nastavovat různé zabezpečovací protokoly. Mohly se připojovat i zařízení třetích stran.

Mnoho informací se daly vyexportovat do CSV souboru. V zabezpečených a víceuživatelských projektech se daly používat certifikáty. Aplikace má zkrátka plno možností.

4.2 Zobrazení informačního textu pro pole

Cílem úkolu bylo zařídit, aby se zobrazoval informační text, který uživateli oznámí, co zapsal špatně nebo co má do textového pole zapsat, aby to bylo vyhodnoceno správně.

Tento úkol nebyl nijak složitý ani náročný. Byl to můj první úkol na praxi, takže jsem nevěděl, kde se jaká třída volá abych mohl text vložit do příslušného boxu. Ovšem základní text, pro

takové problémy tam definovaný byl. Vždycky, když se text nedefinuje, tak je zobrazen text, který informuje, že chybové hlášení doposud nebylo specifikováno.

Našel jsem si tedy, kde se takový text používá. Jakmile jsem jej našel, dal jsem na jeho místo break-point[1]. S nastaveným break-pointem jsem znova zadal špatnou hodnotu a aplikace se mi zastavila. Poté jsem si našel v debuggeru[2], která třída zapříčinila vyvolání takové zprávy.

Vyřešení problému potom již bylo snadné, zjistil jsem si, že třída může překrýt metodu, která vrátí text, jenž bude zobrazen v informačním boxu. Z metody jsem tedy vrátil daný text, vyzkoušel jsem, jestli řešení funguje správně a úkol uzavřel.

4.3 Validace unikátních identifikačních čísel objektů

Každý objekt v aplikaci Acos má své unikátní číslo. Toto číslo je složeno z čísel a písmen abecedy a náhodně generované a vytvořené přes třídu Javy UUID.

Úkol měl dvě části. V první jsem měl najít duplikace nebo null hodnoty těchto id. Druhý úkol byla oprava těchto nežádoucích stavů.

Hlavním problémem bylo vyřešit, jak získat z tak velkého projektu všechna tato čísla. Problém jsem vyřešil použitím třídy, která již existovala a která procházela všechny objekty z daného projektu nebo z daného zařízení. Čísla jsem si uložil do listu, abych je mohl mezi sebou porovnávat a zjišťovat, jestli nemají hodnotu null.

Musel jsem použít vnořené cykly for. Jako první, jsem musel zjistit, jestli se hodnota nerovná null. Pokud se rovnala, vygeneroval jsem náhodné UUID. Když tato podmínka nebyla splněna, měl jsem další podmínku, kde jsem zjišťoval, jestli se v projektu nenachází duplikace.

Tento mechanismus byl složitější, takže jsem pro něj vytvořil novou metodu, která mi vrátila kladnou pravdivostní hodnotu, jestli byla nalezena shoda unikátních čísel. Tato metoda měla v sobě druhý cyklus for a jako parametry přebírala již zmíněný list unikátních čísel a konkrétní číslo, které se kontrolovalo. V cyklu jsem musel zjišťovat, jestli se id vyskytuje v listu více než jednou. Když byla nalezena první shoda, znamenalo to, že jsem porovnával id samo se sebou. Až druhá shoda znamenala, že byla nalezena duplicita a metoda rovnou vrátila výsledek hledání. Objektu s nalezenou shodou se vygenerovalo nové unikátní číslo 1.

[1] Break-point je místo, kde se zastaví aplikace, pokud je spuštěna v debugger
[2] Debugger je softwarový nástroj pro hledání chyb

```
int count = 0;
    for (UUID uuid : listUuid) {
      if (ObjectUtils.equals(uuid.getUuid(), comparedUuid)) {
        count++;
        if (count > 1) {
          return true;
        }
      }
    }
    return false;
```

Výpis 1: Nalezení duplicity unikátního čísla

Výsledek se zobrazoval ve validačním oknu, které bylo rozděleno do tří částí. První část obsahovala výpis chyb, které se automaticky opravily. Zde se tedy objevily i opravené unikátní čísla. Druhá část obsahovala výpis chyb, které může uživatel aplikace sám opravit. V poslední části byly chyby, které už si uživatel sám nemohl opravit a bylo tedy nutné kontaktovat podporu.

Pokud se tedy našla nějaká chyba, uživatel byl nejdřív tázán, jestli si chyby přeje rovnou opravit. Pokud ano, tak se chyby zobrazily v sekci pro automaticky opravené chyby a byl vygenerován textový soubor, který obsahoval výpis všech chyb. Pokud ne, tak se okno s chybami uzavřelo a uživatel pokračoval v práci na projektu.

Tohle byl můj první náročný úkol. Ještě jsem nebyl plně obeznámen se všemi třídami, a tak mi úkol zabral hodně času.

Po dokončení implementace úkolu jsem začal pracovat na JUnit testech. Pro tento úkol jsem nemusel vytvářet úplně novou třídu pro testy, ale musel jsem některé staré testy upravit. Testy po mojí implementaci neprocházeli, jelikož jsem svojí implementací zasáhl i do jiných metod testované třídy.

Jakmile jsem upravil testy, které neprocházely, začal jsem se psaním testů pro můj kód. Při psaní JUnit testů jsem se řídil vzorem 3A (Arrange-Act-Assert) [9]. Takže jsem u každého testu nejdřív připravil všechny objekty a vstupy, které se testovaly nebo byly při testu použity (Arrange). Poté jsem provedl akci na objektu, který se testoval (Act). Nakonec testu jsem porovnával, jestli se výsledek shodoval s očekáváním (Assert). Pokud výsledek nebyl stejný s očekávaným výsledkem, tak se vyhodila výjimka a test neprošel.

Nejdříve jsem otestoval metodu pro korekci unikátních id. Jako první jsem tedy podle vzoru 3A začal vytvářet instance objektů, které jsem potřeboval pro test. Nevytvářel jsem normální instance, ale falešné, pomocí Mockito frameworku. Jakmile jsem vytvořil všechny potřebné objekty, nastavil jsem jim chování, jaké jsem potřeboval.

S vytvořenými objekty, které jsem naučil jak se mají chovat, jsem zavolal testovanou metodu. Metoda mi vrátila výsledek ve formě kolekce Set a ten jsem si uložil. Pomocí metody assertEquals

jsem zjišťoval, jestli se vrácená kolekce rovná kolekci, ve které byly očekávané objekty. Jakmile test prošel, začal jsem psát test pro další metody.

Metoda pro zjištění duplikátních unikátních id byla o dost kratší. Měla méně objektů pro sestavení testu. Vytvořil jsem falešné instance, naučil je chování, jaké jsem chtěl a zavolal metodu, která mi vrátila pravdivostní hodnotu. Pro tuto metodu jsem napsal více testů. Jeden pro případ, kdy metoda nenašla žádné stejné id. Hodnota poslána do metody pro porovnání se v kolekci vůbec nevyskytovala. Další test byl pro nalezení žádné duplikace. To znamenalo, že id poslané do metody v kolekci existovalo, ale vyskytovalo se jenom jednou. Poslední test metody byl pro nalezení duplikace. Takže jsem do metody poslal id, které se v kolekci vyskytovalo více než jednou. Pomocí metody assertTrue nebo assertFalse jsem testy uzavřel a když mi procházely, tak nebyla třeba žádná další úprava.

Pro všechny úpravy kódu a testů jsem vytvořil code-review a úkol byl hotov.

4.4 Otevření projektu ze ZIP archivu

Před tímto úkolem se projekty daly otevřít jenom čistě ze dvou souborů s vlastní koncovkou. Měl jsem naprogramovat, aby se projekt dal otevřít i ze ZIP archivu. Projekty se totiž daly zabalit pouhým stiskem tlačítka v horní části aplikace. Uživatel pak zadal cestu pro uložení archivu a celý projekt i se všemi zařízeními se zabalil. Úkolem tedy bylo, abych naprogramoval funkcionalitu, která zařídí otevírání projektů při vybrání ZIP archivu.

Nejdřív jsem musel najít třídu, ve které se řeší otevírání projektů. Jelikož se archivovat může každý ze tří druhů projektů, tak bylo nutné napsat v místě, kde se ještě neví, jestli je projekt s nebo bez zabezpečení nebo víceuživatelský projekt. Otevírání projektů se řeší jednotně, tak jsem kód umístil na nejvíce vyhovující místo.

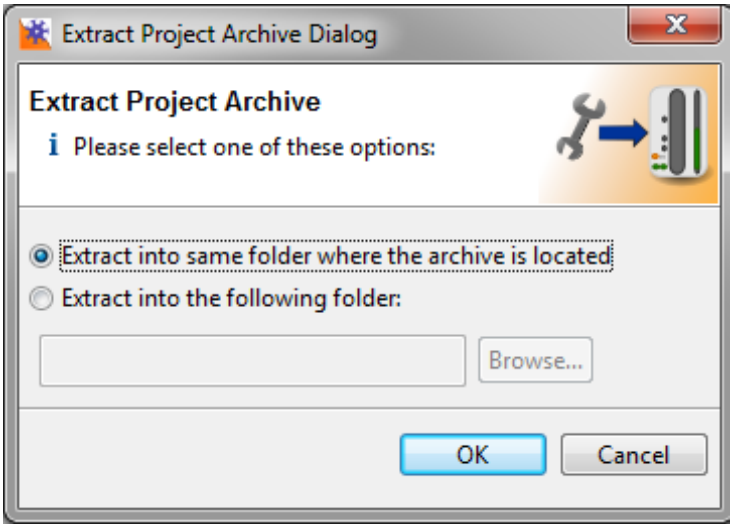
Nejdříve bylo potřeba zjistit, jestli uživatel chce otevřít projekt normálně nebo ze ZIP archivu. K tomu mi posloužila jednoduchá podmínka, kdy jsem se zeptal, jestli název otevíraného souboru končí postfixem .zip. Jiné souborové archivy se v aplikaci nepoužívaly, takže nebyla potřeba kontrolovat postfixy ostatních známých souborových archivů.

Po splnění podmínky bylo uživateli ukázáno dialogové okno, kde měl na výběr ze dvou možností, kam uložit extrahovaný projekt se všemi jeho součástmi. Ukázka dialogového okna lze vidět na obrázku 1 . První volba znamenala, že se archiv rozbalil přímo do adresáře, kde se nacházel. Druhá možnost znamenala, že si uživatel mohl vybrat cestu k adresáři, do kterého chtěl archiv rozbalit.

Co se týče první možnosti, tak to bylo jednoduché. Jelikož už jsem znal cestu k archivu, tak stačilo pouze zjistit rodičovskou složku archivu a výslednou cestu si uložit.

Druhá možnost znamenala, že po stisku tlačítka, které otevřelo okno pro výběr cesty k adresáři, si uživatel zvolil adresář do kterého se obsah archivu extrahoval.

Jakmile si uživatel vybral jednu z možností, tak dialogové okno potvrdil nebo zrušil. Pokud jej zrušil, tak se okno zavřelo a mohl otevřít jiný projekt. Po potvrzení dialogového okna se uložila cesta, kam projekt extrahovat. Mohlo se stát, že by cesta mohla být null a to právě



Obrázek 1: Dialogové okno pro exportování archivu

v případě, kdy uživatel okno uzavřel, takže na místě byla i kontrola na takovou situaci. Jestliže vše bylo v pořádku, tak se cesta k archivu poslala metodě, ve které byl algoritmus pro extrahování archivu.

Musel jsem také zkontrolovat, o jaký typ souboru projektu se jedná. Soubor s projektem může mít totiž dva různé postfixy. Stačilo pouze podmínkou zkontrolovat o který typ se jedná. Kontrola byla nutná, protože metodě, která obsahovala algoritmus pro extrahování archivu jsem musel zadat typ souboru, který se má v archivu vyhledat a ten poté rovnou spustit. V případě, kdy bych měl jeden ze souborů a uživatel by chtěl otevřít druhý typ souboru, tak by se mu vlastně nic neotevřelo, jelikož by metoda žádný takový soubor nenašla.

Výsledek metody byl ten, že archiv rozbalila a jako návratovou hodnotu vrátila cestu k souboru projektu. Ta se poslala jako parametr do již vytvořené metody, která spouštěla projekt a měla v sobě i kontrolu na null hodnotu projektu.

Pro dokončení úkolu jsem tedy vytvořil celkem čtyři nové třídy. Třída ExtractProjectArchive obsahovala vytvoření grafického vzhledu dialogového okna. Pro lepší přehlednost, jsem pro každé tlačítko ve třídě vytvořil samostatnou metodu a jednu metodu, kde jsem poskládal výsledný vzhled.

Okno se skládalo celkem ze dvou kontejnerů. V prvním byly tlačítka pro výběr cesty, kde se má archiv extrahovat a ve druhém bylo textové pole a tlačítko, které spustilo výběr cesty uživatelem.

Při vytváření tlačítka pro extrahování archivu do jeho rodičovské složky jsem naprogramoval posluchače, který v případě zachycení události na tlačítku změnil stavy ostatních komponent. Pokud tlačítko bylo vybrané, ostatní tlačítka pro výběr cesty uživatelem byla vypnuta. Pokud tlačítko nebylo vybráno, uživateli se otevřela možnost zvolit si cestu sám.

Třída také obsahovala další pomocné metody, které budou potřeba v ostatních třídách. Me-

todu pro zjištění, jestli je první tlačítko aktivní, pro vrácení výchozí cesty k archivu a metodu, která vrátí uživatelem zvolenou cestu.

Třída ExtractProjectArchiveBusinessObject přijímala do konstruktoru cestu archivu. Třída dědila z interně vytvořené třídy, takže přepisovala některé metody. Obsahovala také metody pro vrácení cesty, kterou vrací dialogové okno a pro vrácení cesty, která se předala do konstruktoru. Tyto metody se také později využijí v dalších třídách. Tato třída má za úkol v sobě uchovat data, která jsou uživatelem vložena před vytvořením dialogu a po jeho uzavření, jak pomocí potvrzovacího tlačítka, tak pomocí tlačítka pro zrušení okna.

Třída ExtractProjectArchiveDialog také dědí z interní třídy a přijímá do konstruktoru objekt, který obsahuje výše uvedená uživatelem zadaná data. V přepsané metodě se nastavuje okno, které ještě neobsahuje nic, kromě tlačítek pro potvrzení a zrušení dialogového okna. Nastaví se zde také výsledná plocha okna pomoci třídy ExtractProjectArchiveComposite.

Konstruktor přijímá třídu pro uchování dat. Již v konstruktoru třídy se vytváří objekt ExtractProjectArchive. Třída přepisuje dvě metody, které definují chování tlačítek pro potvrzení a zrušení okna.

Pokud je po stisknutí potvrzovacího tlačítka aktivní první výběrové tlačítko, tak se do lokální proměnné uloží výchozí cesta k archivu. Pokud je vybráno druhé výběrové tlačítko, je do této proměnné uložená cesta zadaná uživatelem. Po rozhodnutí, kterou cestu si uživatel zvolil se cesta uloží do objektu pro data.

Po stisknutí tlačítka pro zrušení okna se okno zavře a uloží se prázdná cesta.

Postupnou implementací a nastavení chování těchto mnou vytvořeních objektů se cesta zkontroluje a pokud není prázdná, tak se archiv rozbalí na určené místo.

Tento úkol byl časově náročný, jelikož jeho detaily jsem se dozvídal postupem času.

Pro tento úkol jsem nevytvářel žádnou novou třídu pro testování. Třídy byly již vytvořené a tak stačilo pouze napsat nové testy. První test jsem psal pro metodu, která rozbalila ZIP archiv a vrátila řetězec, který v sobě měl jméno projektu. Nejdřív jsem nastavil, že metoda pro zjištění, jestli je to projekt s běžným postfixem nebo druhým postfixem, vrátí nepravdu, když se bude vyhodnocovat. Poté jsem nastavil, že metoda pro extrahování archivu bude vracet jméno projektu s běžným postfixem. Nakonec se mi potvrdilo, že metoda vrátila co měla a napsal jsem ten samý test, akorát pro projekt s druhým postfixem.

Další testy jsem psal i pro metody, které jsem neimplementoval, ale aby třída byla pokrytá celá testy, tak jsem je napsal i pro tyhle metody. Metodu, která zjišťovala, zda-li se otevírá projekt v archivu nebo ne, jsem pro tuto třídu naprogramoval jako první. Stačilo akorát metodu naučit, jakou má vracet pravdivostní hodnotu a na konci metody otestovat, jestli opravdu vrátí to, co má a test byl hotov.

Zde jsem také testoval metodu, která nic nevrací, jenom provádí algoritmus. To jsem otestoval tak, že jsem si opět připravit všechny objekty a vstupy, které jsem potřeboval. Naučil jsem je chování, jaké jsem potřeboval a potom jsem na testovanou třídu zavolal metodu s objekty

a vstupy. Poté mi akorát stačilo zkontrolovat, že se metoda zavolala s parametry, se kterými měla.

Pro vytvořený kód se všemi testy jsem opět vytvořil code-review pro ostatní programátory a úkol byl hotov.

4.5 Úprava modelu aplikace a transformace projektu

Model aplikace je soubor XML, kde jsou nadefinovány různé atributy a těm jsou přiřazené validátory a kontrolery[3]. Soubor je to velice obsáhlý a zorientovat se v něm zabere hodně času.

Úkol spočíval v tom, abych přidal další objekt do modelu, což bylo jednoduché, protože jsem měl hodně předloh, podle kterých jsem to mohl zrealizovat. Objekt zastupoval IP adresu a síťovou masku.

Nicméně přidáním samotného objektu se problém nevyřešil. Tento objekt totiž měl být v tabulce, která měla obsahovat určitý počet těchto objektů. Tabulka zase měla být v dalším objektu, který ji vlastnil. Tabulku jsem vytvořil tak, že jsem objektu, pod který patřila, přidal referenci na list těchto nových objektů. Bylo také potřeba odstranit některé atributy, které vlastně nahradil objekt, který jsem vytvořil. Musel jsem také naprogramovat různé validace pro nové objekty, ale to je součástí dalšího úkolu, který bude následovat níže.

Po úpravě modelu jsem spustil generátor tříd, který vygeneruje třídy, podle informací v novém objektu. Třídy se vygenerují na určité místo a jsou připraveny k použití. Třídám se také vygenerují metody pro lepší přístup k atributům objektu.

Podle úpravy modelu jsem musel zajistit, že projekty, které byly vytvořeny před změnou, ji budou po transformaci obsahovat. Transformace je vlastně úprava starého projektu do nové verze.

Pro novou transformaci jsem vytvořil třídu, která zajistí potřebný mechanismus. Ve třídě jsem tedy musel nejdřív odstranit a poté přidat nové objekty. Když jsem našel rodiče objektů, nejdřív jsem zavolal metodu pro odstranění objektu a potom pro přidání. Jelikož se na přidaný objekt odkazuje, tak jsem mu musel nastavit, který objekt na něj může mít přístup. Poté jsem objektu nastavil jméno a další informace.

Správnost transformace jsem musel otestovat. Test spočíval v porovnávání dvou XML souborů. Jeden obsahoval staré části modelu i s vyplněnými hodnotami a druhý měl již nové části s hodnotami ze starého modelu. Tyto soubory jsem musel napsat ručně, což ale nebyl žádný problém.

Po úspěšných testech jsem úkol dokončil a začal pracovat na již zmíněných validacích.

4.6 Validace vstupu

Tento úkol následoval hned po úpravě modelu aplikace. Při úpravě modelu se nově přidalo i jedno textové pole pro zadání výchozí brány a další dvě pole, kde se zadávaly adresy pro IP a masku

[3]Kontrolery jsou třídy, které kontrolují stavy objektů v aplikaci

sítě. Pro každé z těchto polí jsem měl naprogramovat kontrolu, že se do polí budou zadávat správné hodnoty.

Úkol tedy byl, aby se do vstupních polí mohly zadávat jenom platné IP adresy, takže jenom hodnoty [0-255].[0-255].[0-255].[0-255] a žádné jiné znaky jako písmena abecedy, speciální znaky a podobně. Dalším požadavkem bylo, aby adresa neobsahovala na začátku hodnotu 10.254. Taky by se neměly v celém projektu vyskytovat stejné IP adresy, pokud ano, tak se mělo vedle pole zobrazit varování ve formě obrázku malého žlutého trojúhelníku s vykřičníkem. Pokud by bylo na vstupu něco jiného, než je povoleno, tak se měla zobrazit chyba. Pro výchozí bránu byla podobná pravidla. Jelikož každé zařízení mohlo mít jeden nebo více ethernet portů, tak bylo pravidlo, že pro každý tento port mohlo být pole výchozí brány prázdné. Další pravidlo bylo, že každý port měl mít jinou bránu. Toto pravidlo mělo výjimku a to takovou, že neplatilo pokud zařízení byla redundantní. Výchozí brána měla být v jednom ze subnetů IP adres nastavených v tabulce nad polem výchozí brány.

Začal jsem tedy s implementací tříd, které pravidla zajišťovaly. Všechny tyto třídy dědily z abstraktní třídy. Pro uplatnění pravidel jsem používal regulární výrazy [3, 4], které když jsem správně napsal, tak zajistily, že uživatel do pole nemohl zadat jiné hodnoty než jaké mohly být.

Každá třída mohla přepsat metodu předka, která jako návratovou hodnotu vrátila všechny validátory, které toto pole mělo obsahovat. Tedy pole pro zadávání IP adresy obsahovalo validátory dva, jeden pro možnost zadání platné IP a druhé pro vynechání IP, která začínají 10.254.

Pro validování unikátních adres v projektu, jsem musel získat všechny IP adresy v projektu, které se týkaly mého úkolu a uložit si je do listu. Definoval jsem tedy servisní třídu, která mi vytáhla z projektu tyto adresy. K tomu sloužila metoda, která měla jako parametr ethernet port, ze které získám všechny jeho IP adresy. Jak už jsem se zmínil, tak zařízení může mít více ethernet portů. Takže jsem musel získat všechny tyto porty z aktuálního zařízení. Když jsem měl všechny tyto údaje, tak jsem použil metodu, pro nalezení unikátních ID z minulého úkolu a nechal jsem ji, ať mi najde, jestli je IP unikátní nebo ne. Pokud nebyla, tak se zobrazilo varování vedle textového pole s textem, který obsahoval popis varování.

Pravidlo, které říkalo, že výchozí brána musí být v subnetu jedné z definovaných IP adres ethernet portů mělo podobné řešení. Získal jsem si port, ve kterém výchozí brána byla nastavená a ze servisní třídy jsem si vytáhl všechny jeho použité adresy. Z další servisní třídy jsem zjistil, jestli se výchozí brána vyskytuje v rozmezí subnetu některé z IP a pokud ne, tak jsem nechal zobrazit chybový obrázek s popisem chyby.

Všem popisům chyb a varováním jsem nastavil informace o problémech. K tomu stačilo jenom přepsat další metodu předka a vrátit textovou zprávu s popisem chyby či varování.

Musel jsem taky vytvořit servisní třídu pro ethernet port, která po transformaci zkontroluje, jestli tabulka má správný počet IP adres. Byl totiž požadavek, že v tabulce musí být minimálně jedna a maximálně šedesát čtyři adres.

Pro každý tento chybový stav jsem vytvořil metodu, která v případě výskytu chyby vyvolá výjimku, která se zachytávala v jiné třídě pro výpis chyb.

Pro případ, kdy není žádná adresa v tabulce jsem napsal podmínku, která se splnila v případě, kdy list IP adres je prázdný. Naopak druhá metoda kontrolovala, jestli v listu je více jak určitý počet adres. Pokud jich bylo více, vyvolala se výjimka.

Jakmile jsem všechno odzkoušel přímo v aplikaci a funkcionalita byla podle zadání, pustil jsem se do testování kódu.

Testování pro vytvořenou servisní třídu a validátory nebylo nijak obtížné. Musel jsem vytvořit testovací třídy a v nich provádět všechny testy.

V případě servisní třídy jsem do metody, která se před každým testem spustí, vytvořil novou instanci třídy a nastavil jsem všechny mocky[4], které jsem definoval na začátku třídy. Poté jsem vytvořil test, který testoval, že se nevyhodí výjimka v případě výskytu chyby. To jsem zařídil tak, že jsem do bloku try/catch vložil kód, který spustí metodu. Níže jsem potom vložil řádek, který vyhodí chybu, že se výjimka nevyhodila. To jsem zařídil pomocí metody fail() a do argumentu jsem napsal důvod, proč se metoda zavolala. V catch bloku jsem zachytával výjimku. Takže pokud se výjimka nezachytla, vypsala se hláška v metodě fail() zobrazené ve výpisu číslo 2.

```
@Test
  public void test() throws Exception {
       try {
      target.doSomething(arg1, arg2);
      fail("ChildOfException expected");
    } catch (ChildOfException e) {
    }
  }
```

Výpis 2: JUnit test

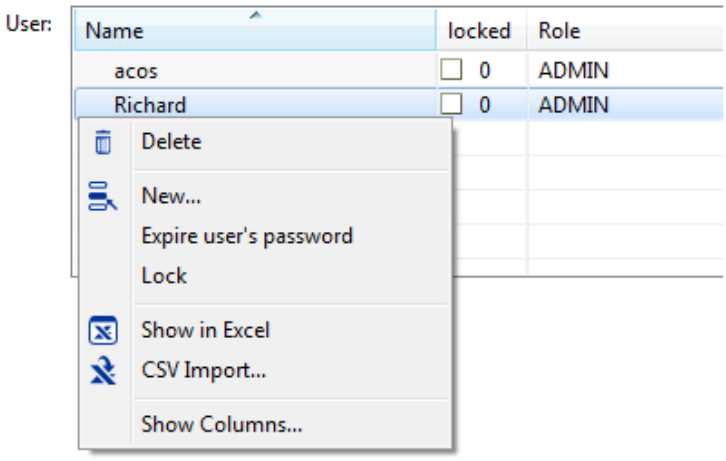
Pro chybu s překročením maximálního počtu adres jsem udělal podobný test. Místo prázdného listu jsem naučil objekt vracet list s překročeným počtem adres. Postup byl jinak stejný jako test pro prázdný list.

Pro testy validátorů jsme používali abstraktní třídu, kterou vždycky rozšiřuje test pro konkrétní validátor. Je to proto, že pro každý validátor se dělaly stejné základní testy. Takže test pro konkrétní validátor akorát přepsal metody a vracel potom hodnoty, které potřeboval pro svůj test.

V případě validátoru pro výchozí bránu jsem implementoval testy, které kontrolovaly, že se zvalidují správné statusy. Statusy mohly být tři: status pro úspěch, chybu nebo varování. V tomto validátoru se nekontroluje žádný status pro varování, jenom pro úspěch a chybu.

První test byl pro prázdnou výchozí bránu. Do proměnné pro status jsem si uložil výsledek validace s hodnotou null nebo prázdným řetězcem. Potom jsem v metodě předka porovnal výsledky a očekávaný status. Pokud se hodnoty shodovaly tak test prošel.

[4]mock je falešná instance objektu. Vytváří se pomocí Mockito frameworku



Obrázek 2: Kontextové menu

O něco složitější byl test pro zjištění, jestli je výchozí brána v některém se subnetu uložených lokálních adres. Test jsem psal pro výchozí bránu v subnetu i pro případ, kdy se brána v žádném ze subnetů nevyskytovala.

V prvé řadě jsem musel zařídit, aby ethernet port měl nějaké adresy. Proto jsem vytvořil privátní metodu, která naplňovala falešné IP adresy. Poté jsem naučil metodu, která zjišťovala jestli je IP prvního parametru ve stejném subnetu jako IP v druhém parametru, ať má vracet hodnotu pravda, když se zavolá. Do proměnné jsem si, jako další krok, uložil výsledný status validace. Pak již zbývalo jenom porovnat výsledný status se statusem očekávaným a test byl hotov.

Ten samý postup jsem aplikoval i pro případ, kdy brána nebyla v žádném ze subnetů.

4.7 Přidání akce do kontextového menu

Ve více uživatelském projektu se nachází také sekce, kde jsou zobrazeni všichni uživatelé, kteří mohou k tomuto projektu přistupovat a různě ho měnit. Je více rolí pro uživatele, základní jsou administrátor a operátor. Administrátor může dělat s projektem vše, přidávat uživatele, certifikáty a tak dále. Operátor má některé funkce zakázané, například přidávání certifikátů a uživatelů.

Uživatelé jsou vypsaní v tabulce, která obsahuje uživatelské jméno, roli uživatele a jestli je zamknutý nebo ne. Zámek lze vidět v tabulce jako zaškrtávací políčko. Dá se zapnout nebo vypnout v kontextovém menu, když administrátor klikne pravým tlačítkem myši na uživatele. Poté vybere funkci Lock/Unlock podle toho, jestli je uživatel zamknutý nebo ne. Ukázku kontextového menu lze vidět na obrázku 2.

Jako první jsem vytvořil třídu, která dědila z předka. Tento předek zajišťoval funkce, které byly potřeba. Přepsal jsem jednu metodu, která zajišťovala spouštění akce.

Potřeboval jsem vytvořit metodu, která zajišťuje změnu popisku zámku. Například, když je uživatel zamknutý a administrátor spustí akci, aby se text změnil na „Unlock" odemknout. Jakou hodnotu zámku uživatel má, šlo zjistit, když jsem si jej z tabulky získal a poté jsem na jeho objekt zavolal metodu pro zjištění stavu zámku. Metoda vrátila pravdivostní hodnotu podle stavu. Tedy pravda, pokud je uživatel zamknutý a nepravda, pokud není. Podle toho jaký byl zjištěný stav, tak se nastavil text opačného stavu do kontextového menu a změnil se stav zámku uživatele.

Další metoda, kterou jsem vytvořil, sloužila pro uložení projektu. Projekt jsem potřeboval uložit na disk a také na SVN server. K tomu sloužila opět další servisní třída, která tyto akce zajišťovala. Stačilo tedy zavolat metodu ze servisní třídy pro uložení projektu a metoda byla hotová.

Jádro celé funkcionality bylo naprogramované v překryté metodě předka. Všechno to bylo zabalené do bloku try/catch, protože každá manipulace s SVN projektem může vyhodit výjimku. Nejdřív ze všeho jsem musel zamknout celý projekt, aby byla zajištěna konzistence dat. Potom jsem zavolal metodu pro zamykání uživatele. Hned poté jsem projekt uložil pomocí SVN servisní třídy a odemkl pro případné další změny.

Celou třídu jsem otestoval pomocí JUnit testů a také ji prozkoušel v aplikaci. Všechno fungovalo podle zadání, tak jsem úkol uzavřel.

4.8 Vložení ukazatele postupu algoritmu

Když se v aplikaci načítá velký projekt a je jedno jestli je to zabezpečený, normální nebo víceuživatelský, tak se může stát, že aplikace zamrzne a nic se s ní nedá dělat. Pozná se to tak, že se v hlavičce aplikace zobrazí text „Neodpovídá" a okno s aplikací zbělá. Tohoto jevu jsem se měl zbavit.

Podobných úkolů jsem měl více a všechny měly stejný postup. Nejdříve jsem musel najít místo v kódu, které způsobuje onen nežádoucí stav. Samozřejmě jsem věděl, při jaké akci se to stane. První takový úkol, který jsem měl vyřešit nastal po stisknutí konkrétního tlačítka. Takže jsem si aplikaci spustil v debuggeru a zastavil si ji tam, kde jsem si myslel, že nastane daný stav. Jakmile jsem zjistil, která metoda nebo který algoritmus způsobuje tento nežádoucí jev, vložil jsem jej do metody „run" monitoru [8], který zobrazuje stav dokončení algoritmu. Monitor je součástí třídy Javy a tak jsem jej použil jako v ukázce číslo 3.

```
PlatformUI.getWorkbench().getProgressService().busyCursorWhile(new
    IRunnableWithProgress() {

        @Override
        public void run(IProgressMonitor monitor) throws InterruptedException {

         }
);
```

Výpis 3: Ukázka kódu pro grafický ukazatel postupu

Jakmile jsem kód obalil do monitoru, vyzkoušel jsem jeho funkcionalitu, jestli se už zamrzávání neobjevuje a úkol byl hotov.

4.9 Kontrola víceuživatelského projektu

Při vytváření víceuživatelského projektu, se zkontroluje zadaná URL adresa SVN serveru. To znamená, že se zkontroluje, jestli zadaná URL existuje. Jakmile se provede úspěšná kontrola, projekt se musí uložit. Na určené místo se potom uloží i skrytá složka, která obsahuje informace o víceuživatelském projektu.

Úkolem bylo zajistit, aby se uživatel nedostal do projektu, pokud tato skrytá složka neexistuje a aby se zobrazila chybová hláška.

K vyřešení úkolu mi postačilo znát servisní třídu pro víceuživatelské projekty a tam se podívat, jestli nenajdu metody, které by mi mohly pomoct. Našel jsem dvě metody, kde jedna kontroluje, jestli je v projektu URL SVN serveru zadaná. Druhá metoda kontroluje, zda-li složka, ve které se projekt nachází obsahuje skrytou podsložku pro SVN projekty.

Pro úspěšné vyřešení úkolu jsem nejdříve musel zkontrolovat, jestli projekt obsahuje URL. Pokud by totiž neobsahoval, nejednalo by se o více uživatelský projekt. Pokud kontrola zaznamenala, že adresu projekt obsahuje, přešlo se na další kontrolu, jestli složka s projektem obsahuje skrytou SVN složku. Pokud neobsahovala, znamenalo by to, že byla buď úmyslně odstraněna a jednalo by se o nežádoucí pokus o vstup do projektu, nebo se nějakým nedopatřením uživateli povedlo složku odstranit.

Po krátkém otestování v aplikaci jsem zjistil, že algoritmus funguje a úkol jsem mohl uzavřít.

4.10 Vypnutí komponent, které nejsou relevantní pro Windows autentizaci

V chráněném projektu lze nastavit sílu hesla. To znamená, že administrátor může nastavit, jakou minimální délku heslo musí mít, kolik malých a velkých písmen, čísel a speciálních znaků. Heslu v aplikaci lze také nastavit, po jaké dlouhé době si uživatel musí heslo změnit. Všechny tyto hodnoty se zadávají do textových polí.

Úkol byl takový, že pokud projekt byl víceuživatelský, musela se tato textová pole vypnout, tzn. aby do nich nešly zadávat hodnoty. K tomu jsem využil kontrolery. Musel jsem přepsat rodičovskou metodu, která zajišťuje vypnutí pole. Ta vrací pravdivostní hodnotu podle toho, jestli má být vypnutá.

Pole se mají vypnout, pokud je projekt víceuživatelský. To jsem zjistil pomocí servisní třídy, která obsahuje metodu pro zjištění objektu. Metoda přijímá jako jediný parametr projekt. Ten jsem získal z rodiče kontroleru, protože obsahuje metodu, která vrací aktuální projekt.

Jakmile jsem funkcionalitu odzkoušel, úkol jsem dokončil a uzavřel.

4.11 Viditelnost sekce po zaškrtnutí tlačítka

Měl jsem za úkol zajistit, aby sekce v zařízení byla neviditelná po zaškrtnutí tlačítka, které patří jiné sekci. Nabízela se mi možnost vypnutí celé sekce.

Abych možnost mohl uskutečnit, potřeboval jsem vyvolat událost, která se zveřejní po zaškrtnutí tlačítka. To jsem zařídil v kontroleru tlačítka. Kontroler má totiž metodu, ve které mohou být algoritmy, které mají proběhnout při každé interakci tlačítka. Takže jsem si nejdříve vytvořil třídu, která mi zastupovala onu událost. Třída neobsahovala vůbec nic ani konstruktor. Potřeboval jsem ji pouze zachytit v kontroleru sekce, která se má skrýt. Událost jsem vyvolal pomocí interní servisní třídy a její metody. Do metody jsem dal novou instanci události a událost byla tím pádem vyvolána po každé změně stavu tlačítka.

Dále jsem si našel kontroler sekce, kterou potřebuji zobrazovat v závislosti na stavu tlačítka. V kontroleru jsem zachytával onu vyvolanou událost a když jsem ji zachytil, tak jsem sekci chtěl změnit stav. Stav viditelnosti se dal jednoduše změnit pomocí metody předka, ze kterého kontroler dědí. Po zachytnutí události jsem musel zjistit, v jaké stavu je tlačítko, které událost vyvolalo. K tomu mi opět posloužila interní servisní třída, která měla metodu, pro zjištění stavu tlačítka. Použil jsem ji v podmínce a pokud byla pravdivá, tak jsem zařídil, aby sekce byla neviditelná.

Po otestování byla funkčnost správná, takže by se úkol dal uzavřít. Problém ovšem nastal tehdy, když jsem otevřel sekci, která se nacházela přímo pod upravovanou sekcí. Její název se změnil na název neviditelné sekce a nedala se potom otevřít, protože si aplikace myslela, že chci otevírat tu neviditelnou sekci. Dále byl problém, že se ostatní sekce nesprávně umisťovaly v editoru v závislosti na viditelnosti sekce. Po diskuzi s německým kolegou jsem zjistil, že aplikace není schopna dynamicky skrýt a znovu objevit jakoukoliv sekci, takže jsem implementaci musel zrušit a přistoupit k další možnosti řešení úkolu.

Ta spočívala v tom, že pro každou komponentu v sekci jsem nastavil, že má být neviditelná po zaškrtnutí tlačítka. To jsem implementoval v kontrolerech jednotlivých komponent. Některé komponenty, které měly i svůj vlastní validátor, musely mít nastavené, aby se nevalidovaly pokud tlačítko bude zaškrtnuté. Pokud totiž tyto komponenty budou vypnuté, není třeba kontrolovat jejich vstupy.

Jelikož by se podmínka pro zjištění stavu zaškrtnutého tlačítka opakovala v každém kontroleru nebo validátoru komponenty, vytvořil jsem servisní třídu, která obsahovala jedinou metodu pro zjištění stavu tlačítka. Metoda přebírala jeden parametr a to společného předka pro všechna zařízení v aplikaci. Podmínka se má totiž vyhodnotit pouze pro jedno určité zařízení. Metoda tedy obsahovala if podmínku, která vyhodnotila jestli je zařízení správné a poté vrátilo pravdivostní hodnotu stavu tlačítka. Jestli je zaškrtnuté tak pravda, jestli ne tak nepravda.

Servisní třídu jsem vytvořil a musel jsem ji ještě namapovat, aby se k ní všechny kontrolery a validátory dostaly. To jsem zařídil v servisní třídě pro mapování nově vytvořených servisních tříd.

Každý kontroler mohl přepisovat metodu předka, která zajistí, že je komponenta vypnutá nebo zapnutá. Pokud má být vypnutá, vrací se pravdivostní hodnota nepravda a pokud zapnutá tak pravda. Některé komponenty už měly vytvořený kontroler, tak k těm jsem akorát do metody pro zapnutí a vypnutí přidal podmínku, kterou jsem implementoval v servisní třídě. Pro ty komponenty, které neměly vlastní kontroler, jsem musel vytvořit nový a stačilo abych přepsal metodu předka a vložil do ní metodu servisní třídy a bylo hotovo.

Validátor všechny komponenty mít nemusely. Pro ty, které již měly, jsem opět přepsal metodu předka, která zajišťovala, že se komponenta nebude validovat. Poté stačilo znova do metody implementovat metodu servisní třídy a bylo hotovo.

Po testování funkčnosti vše běželo, jak bylo v úkolu zadáno a začal jsem psát testy pro všechny nově vytvořené třídy. Pro ty, které testy již měly, jsem musel testy upravit, aby procházely po nové úpravě chování.

První test jsem vytvořil pro metodu servisní třídy. Jednalo se akorát o naučení metody, aby vracela pravdivostní hodnotu, kterou jsem potřeboval a poté stačilo jenom otestovat pomocí metody assertTrue(), že testovaná metoda vrátí hodnotu true.

Další na řadu přišly validátory. U těch jsem otestoval, že pokud dostali hodnotu, která se měla validovat jako nesprávná, vrátili mi chybový status a naopak. Pro tyhle testy jsme měli v abstraktní třídě metody, které přijímaly dva parametry. První byl testovaná třída a druhý hodnota pro validaci. Takové metody jsme měli dvě. Jednu pro validaci správných hodnot, ta mi vrátila status, že je vše v pořádku. Druhá byla s chybovým statusem. Testy jsem napsal pro správné i nesprávné vstupy.

Testování kontrolerů probíhalo obdobně. Naučil jsem metody chování, jaké jsem potřeboval a poté jenom kontroloval výstup pomocí metod assertTrue() nebo assertFalse().

5 Použité a chybějící znalosti

5.1 Využité znalosti

Ze školy jsem toho na praxi použil mnoho. Hlavně co se týče programování. Využil jsem znalosti ze Základů programování, které již neexistuje a místo toho se předmět rozdělil na dva předměty v prvním ročníku. Dále jsem využil znalosti z předmětu Programovací jazyky I. Tam jsem se seznámil se základy jazyka Java. Z předmětu Vývoj informačních systému jsem věděl, že existují návrhové vzory. Jeden z nich, State[1], jsme implementovali do aplikace. Závěrečný projekt jsem dělal v programovacím jazyce Java, takže i tady jsem získával zkušenosti s tímto jazykem.

Objektově orientované programování mě provázelo po celou dobu studia, proto tyto zkušenosti a znalosti jsem mohl také aplikovat v praxi.

5.2 Chybějící znalosti

S čím jsem se ve škole nesetkal a dost jsem potřeboval na praxi bylo testování pomocí JUnit. Takže jsem se musel naučit všechny jeho postupy sám nebo jsem dostával rady od kolegů. Po zkušenosti z praxe si myslím, že testování by mělo být součástí každého napsaného kódu neboť může odhalit chyby v algoritmu dříve, než při běhu aplikace..

6 Závěr

Při nástupu do firmy jsem měl obavy, že mi nebudou mé odborné znalosti stačit. Ovšem dostal jsem se do týmu, který měl velké pochopení pro mé případné neznalosti, jako třeba v případě JUnit testování. Celý čas jsem se věnoval programování v jazyce Java, což byl můj cíl. Prohloubil jsem si znalosti programování v tomto jazyce a naučil se nové frameworky.

Jelikož jsem pracoval v týmu, tak jsem poznal, jak se rozděluje práce pro více lidí. Jaké to má výhody a jaké to může mít i nevýhody. Hlavní nevýhodou je, že se programátoři někdy neshodnou na určitě věci a potom se práce zpozdí, než se najde kompromis.

Ve firmě TDS jsme měli také jednu hodinu týdně konverzaci s rodilým mluvčím angličtiny. Byl to pro mne velký benefit, který se určitě bude hodit v každé mezinárodní společnosti.

Praxe pro mě byla dobrou profesní zkušeností, jelikož jsem poznal, co všechno obnáší programování aplikace. Proto bych klidně volil bakalářskou praxi znovu.

Literatura

[1] PECINOVSKÝ, Rudolf. *Návrhové vzory: 33 vzorových postupů pro objektové programování.* Vyd. 1. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[2] SCHILDT, Herbert. *Mistrovství - Java.* 1. vyd. Brno: Computer Press, 2014. Mistrovství. ISBN 978-80-251-4145-8.

[3] *Java Regex - Tutorial* [online]. 2015 [cit. 8-1-2016]. Dostupné z: http://www.vogella.com/tutorials/JavaRegularExpressions/article.html

[4] *Shrnutí syntaxe regulárních výrazů* [online]. 2014 [cit. 8-1-2016]. Dostupné z: http://www.regularnivyrazy.info/shrnuti-syntaxe.html

[5] *Historie firmy TDS* [online]. 2010 [cit. 11-1-2016]. Dostupné z: http://www.tds.cz/index.php?lang=cs&page=history

[6] *Eclipse RCP (Rich Client Platform) - Tutorial* [online]. 2016 [cit. 13-1-2016]. Dostupné z: http://www.vogella.com/tutorials/EclipseRCP/article.html

[7] *JUnit* [online]. 2016 [cit. 13-1-2016]. Dostupné z: http://junit.org

[8] *How to Correctly and Uniformly Use Progress Monitors* [online]. 2006 [cit. 3-3-2016]. Dostupné z: https://eclipse.org/articles/Article-Progress-Monitors/article.html

[9] *Arange Act Assert* [online]. 2012 [cit. 14-3-2016]. Dostupné z: http://c2.com/cgi/wiki?ArrangeActAssert